Aspect-Oriented Instrumentation for Capturing Task-Based Event Traces

Yonglei Tao
School of Computing and Information Systems
Grand Valley State University, Allendale, MI 49401, USA
Email: taoy@gvsu.edu

***Abstract* - User interface events provide valuable information about user behavior with respect to an application's user interface and therefore are considered as an important source of data for usability evaluation. Since usability evaluation is based on tasks that users perform, it is crucial to capture user interface events with sufficient information for task identification. However, how to make it possible is still an open question. We describe an approach to investigate the use of the aspect-oriented technique with assistance of localized instrumentation to capture task-based event traces. We also present a proof-of-concept tool to demonstrate its effectiveness and discuss its limitations.**

***Index Terms* - Software Engineering, Tool Support, Aspect-Oriented Programming**

I. Introduction

User interface development is an exploratory and evolutionary process, driven largely by early and ongoing usability evaluation [11]. Changes in the user interface of a window-based application inevitably affect other components in the application. Evolution of window-based applications imposes a challenge on tool support for usability evaluation.

User interface events occur in a window-based application and its run-time environment when the user interacts with the application via its user interface. Since such events provide valuable information about user behavior with respect to the user interface, they are considered as an important source of data for usability evaluation [1]. A range of techniques are available for tracing user interface events. Instrumentation requires that the developer write code in specific locations in the target application to capture the needed information during execution. Instrumentation code tends to be distributed throughout the application and therefore is especially inflexible with an evolving application [4, 5]. Recent work focuses on Aspect-Oriented Programming (AOP) [3-8]. AOP offers an effective way to modularize crosscutting concerns in an application, allowing the developer to gather information from an application in a non-intrusive manner [9]. As used for event tracing, it is highly adaptable, but does not always obtain as much information as needed.

Usability evaluation is to find out to what extent a user interface allows users to accomplish their tasks effectively and efficiently. In order to support usability evaluation, it is crucial to possess the ability to identify tasks and task sequences from event traces [2]. In most cases, task identification requires contextual information, but how to extract the relevant context along with event traces remains an open question [1, 3]. In this paper, we describe an approach to investigate the use of the aspect-oriented technique with assistance of localized instrumentation for capturing task-based event traces.

The rest of this paper is organized as follows. Section 2 identifies some problems with the lack of contextual information for a meaningful interpretation of event traces. Section 3 describes an approach that allows relevant information to be taken from an application to facilitate task identification. Section 4 shows the structure of a proof-of-concept tool that is built according to this approach and section 5 discusses its effectiveness and limitations. Finally section 6 concludes this paper.

II. Contextual Information

AspectJ is an aspect-oriented extension to the Java programming language [9]. It provides syntactic constructs to specify crosscutting concerns. In AsepctJ, an aspect is composed of pointcuts and advices. A pointcut declares a set of well-defined points, referred to as join points, such as a call to a method in an application. An advice is a piece of code that defines actions at the advised joint points, for example, gathering certain information about those joint points. Once an application is compiled with one or more aspects, additional behavior is integrated into the target application.

```
public aspect ActionEventTraceAspect
{
  pointcut actionEventPointCut (ActionEvent e)
    : execution (void ActionListener+.actionPerformed(ActionEvent))
    && args(e);

  before(ActionEvent e): actionEventPointCut (e)        // an advice
  {
    System.out.println (e.getSource().getClass().getName() + " "
                        + e.getActionCommand() + " Clicked ");
  }
}
```

Figure 1: An Aspect for Tracing Action Event

Figure 1 shows the definition of an aspect that is intended to identify the source of an action event when the actionPerformed() method of an action listener is executing. If the user clicks the "Deposit" button in the user interface, for example, it will display a string of "JButton Deposit Clicked". Since its definition is application-independent, this aspect can be used to trace action events for any Java application.

DOI: 01.IJCSI.03.01. 41

Likewise, aspects can be defined for tracing other types of events. Using these aspects makes it possible to collect event traces that result from steps taken by users in completing their tasks. Here, the question is whether or not the collected data is sufficient for indentifying user tasks.

We use a window-based application in Java, called AccountManager, as an example for further discussion. Briefly, this application allows the user, that is, a bank customer, to manage each of his/her bank accounts in a separate window. In the window for an account, it displays the account balance as a text and as a bar graph, and also includes a text field and two buttons, where the former allows the user to enter an amount and the latter to withdraw and deposit the input amount, respectively. In addition, this application shows the total assets held in all of the customer's accounts as a pie chart.

```
JTextField Gained Focus
Key 1 Pressed in JTextField
Key 0 Pressed in JTextField
Key 0 Pressed in JTextField
JButton Deposit Clicked
```

Figure 2: Sample Event Trace from One Window

Given in Figure 2 is an event trace captured from the AccountManager application by using the above-mentioned aspects. We have removed timestamps and irrelevant information from the raw data. Note that the only application-specific element in the sample data is the word "Deposit"; it is a string that is associated with a command button and extractable from an action event. But such a string is not available to some other screen components, such as text fields.

According to the sample data in Figure 2, we know that the user was trying to make a deposit in an account; however, we do not know which account he/she intended for. It is difficult, if not impossible, to give a meaningful interpretation to the collected data. Such a problem becomes more obvious when the user works with more than one window at the same time.

Figure 3 shows an event trace that originates from user actions on two windows; here, events from two different tasks are interleaved. In the AccountManager application, a window is created for an account and different windows are different instances of the same window class. In this case, tracing the current window is not particularly helpful either. Clearly, events as such do not carry enough information on their own to allow their meaning to be properly interpreted. If we cannot relate individual events according to what they are intended for, we won't be able to identify user tasks and task sequences correctly.

AspectJ also offers the ability to extract information (i.e., class names and memory addresses) about the executing and affected objects at the advised joint points. But, it still falls short of what is needed for task identification.

```
JTextField Gained Focus
Key 1 Pressed in JTextField
Key 0 Pressed in JTextField
Key 0 Pressed in JTextField
JTextField Gained Focus
Key 2 Pressed in JTextField
Key 5 Pressed in JTextField
Key 0 Pressed in JTextField
JButton Deposit Clicked
Dialog Confirmation returned YES_OPTION
AssetPieChartView Updated
AccountBarGraphView Updated
AssetTextView Updated
JButton Deposit Clicked
Dialog Confirmation returned YES_OPTION
AssetPieChartView Updated
AccountBarGraphView Updated
AssetTextView Updated
```

Figure 3: Sample Event Trace from Two Windows

III. Our Approach

As discussed above, task identification requires additional information beyond what the event tracing aspects are able to capture in an application-independent manner. Some researchers propose to align data obtained by other methods, such as a video recording, with the automatically collected data through timestamps to gain the contextual information [5, 10].

```
private class FocusHandler extends FocusAdapter
{
   String activeComponent;

   public void focusGained(FocusEvent e)
   {
      if (e.getSource() == amountTextField)
      {
            amountTextField.selectAll();
            activeComponent = new String("Textfield Amount");
      }
      else ...          // other cases are omitted
   }

   public String toString ()
   {
      return activeComponent + " for " + getAccount().getName();
   }
}
```

Figure 4: Added Code in the Focus Listener

Our approach is to allow the application developer to augment certain elements of the target application, making the needed information extractable by the event tracing aspects. Whereas the needed information varies from application to application, the locations where code modifications take place are common to all of them. Consequently, application-specific modifications are localized and can be done with limited effort and in a consistent manner.

Figure 4 shows a listener class for the focus event. It defines the focusGained() method to describe actions to be taken when the user moves the focus to a screen component.

DOI: 01.IJCSI.03.01.41

Unlike a typical listener, it contains extra lines of code (i.e., those with gray background) that enables the toString() method to return information about the active component and the intended bank account.

```
public aspect FocusEventTraceAspect extend AbstractTraceAspect
{
  pointcut captureFocusEvent ()
    : execution (void FocusListener+.focusGained(..));

  before(): captureFocusEvent ()
  {
    report (thisJoinPoint.getThis() + " GainedFocus");
  }
}
```

Figure 5: An Aspect for Tracing Focus Events

Figure 5 shows an aspect that describes what information to capture when the FocusGained() method is executing on a FocusHandler object. Here, the report() method, inherited from AbstractTraceAspect, is responsible for accumulating the collected data; its parameter includes the "this" object, implying a call to the toStirng() method for the FocusHandler object. As such, this aspect can be used to detect the component that has received the focus.

```
Textfield Amount for Account 2 Gained Focus
Key 2 Pressed in Textfield Amount for Account 2
Key 0 Pressed in Textfield Amount for Account 2
Key 0 Pressed in Textfield Amount for Account 2
Button Withdraw in window for Account 2 Clicked
Dialog Confirmation with window for Account 2
returned YES_OPTION
Asset Pie Chart View for All Accounts Updated
Bar Graph View for Account 2 Updated
Asset Text View for Account 2 Updated
```

Figure 6: Sample Event Trace Resulting from Our Approach

Figure 6 shows an event trace obtained from the AccountManager application according to this approach. As we can see, inclusion of the account information makes it possible to obtain a better understanding on user actions and also to relate individual events on the basis of their common task. In case of interleaved events due to multi-tasking, it is still possible to distinguish event traces on the basis of tasks.

In Java, every class inherits the toString() method from the common ancestor class Object. If the FocusHandler class does not override its toString() method, the FocusEventTraceAspect would capture the information provided by the default implementation of toString(), that is, the hashcode of the current FocusHandler object in the form of class-name@memory-address; although a hashcode is unique for every object, it is insufficient to serve for task identification.

IV. A Proof-Of-Concept Tool

User interface events are generated at various levels as natural products of the normal operations of a window-based application [1]. We developed a proof-of-concept tool to trace user interface events on the basis of our approach. Many existing tools take aim at low-level events, but our focus is on events at the application level with the intent to facilitate task identification. Low-level events are extremely voluminous and rich in detail, whereas application-level events are more amenable to task-based analysis.

Java IDE's (Integrated Development Environments) provide powerful support for user interface implementation. For example, NetBeans allows the developer to select an event that one wants to handle from an event list and then specify the actions in an automatically generated method stub [12]. Use of an IDE imposes considerable uniformity on structural characteristics for the application under development; hence, making it more attractive to focus on application-level events.

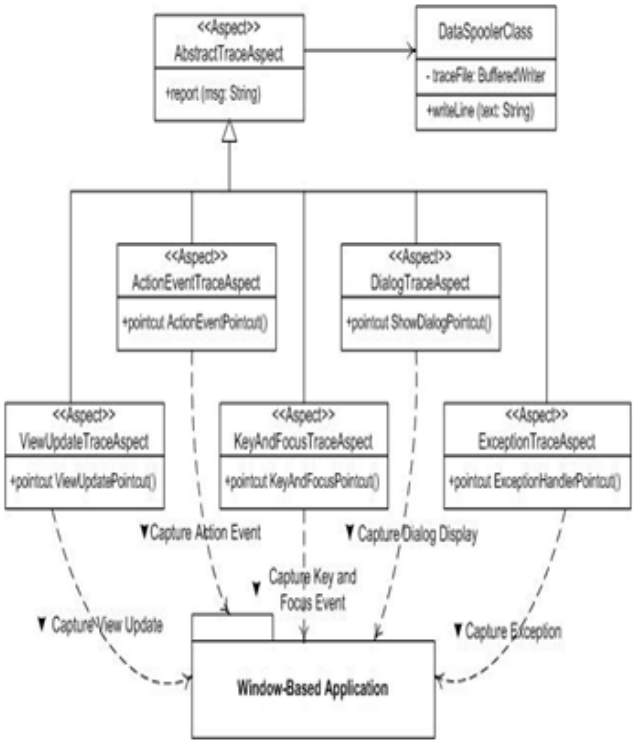

Figure 7: Structure of the Proof-Of-Concept Tool

Figure 7 shows the basic structure of this tool. Aspect AbstractTraceAspect and class DataSpoolerClass encapsulate common event-tracing behavior, such as attaching a timestamp to each event, accumulating the collected data, and generating a report. Other aspects are defined to capture different types of events. Among these aspects, ActionEventTraceAspect and KeyAndFocusTraceAspect are to trace events that are triggered by user actions, and ViewUpdateTraceAspect, DialogTraceAspect, and ExceptionTraceAspect to trace events that are generated by the target application in response to user actions.

As a matter of fact, the sample data shown in Figure 6 is captured from the AccountManager application by using this tool. Due to limited space, we omit detailed information about this tool.

DOI: 01.IJCSI.03.01.41

V. Discussion

A fundamental problem with collecting event traces without contextual information is that developers have no way to answer the "why" behind the "what" when analyzing the collected data [10]. However, there are no guidelines that developers can use to specify the relevant context in their applications [3]. In this paper, we restrict ourselves to what is needed to identify tasks and task sequences.

As reported in the literature, it seems to be inevitable to involve the relevant context in the usability evaluation process. Some proposals require in-depth knowledge about the target application in order to define join points in the event tracing aspects [6-8] and others make use of application-specific information when post-processing the collected data [3-5]. As we can see, the former asks the application developer to modify portions of the tool one uses and the latter asks the tool developer to make adjustments to the tool according to the target application. In either case, they need to have sufficient knowledge about the code that other people create and they might not be familiar with. In contrast, our approach only asks the application developer to modify portions of their own code. Moreover, usability evaluation is a joint effort of the user and the application developer; it is in the developer's interest to obtain sufficient usability data and find out problematic areas for improvement. Hence, the provision for the developer's involvement is not inappropriate.

A potential limitation of our approach is its inability to capture some events that the target application is not intended to respond but are still valuable for usability evaluation [3]. Consider a scenario that the user selected several menus by mouse clicks without taking any action. Such a scenario may indicate that the user had some difficulty with deciding what to do next. One possible solution is to trace selected low-level events in addition to those at the application level.

One issue with our approach is how to deal with the evaluation-oriented code added by the developer when the target application is ready to use. Since there is no simple way to release the added code from the target application, tool support for adding and removing code is necessary. On the other hand, if usability evaluation is an on-going activity, even in the operational stage, it is important to make sure that the added code won't interfere with the normal use of the target application.

VI. Summary

Our approach uses aspects to modularize the event tracing capability and instrumentation to make contextual information extractable. Using the event tracing aspects with assistance of localized instrumentation paves the way for capturing task-based event traces. However, this approach is not a complete one without a suitable solution for data analysis. Further research is underway.

Reference

[1] D. M. Hilbert and D. F. Redmiles, "Extracting Usability Information from User Interface Events," ACM Computing Surveys, Vol. 32, No. 4, pp. 384-421, Dec. 2000.
[2] M. Ivory and M. Hearst, "The State of the Art in Automating Usability Evaluation of User Interfaces," ACM Computing Survey, Vol. 33, No. 4, pp. 470-516, Dec. 2001.
[3] G. Hartman and L. Bass, "Logging Events Crossing Architectural Boundaries," Proceedings of the 10th IFIP TC13 International Conference on Human-Computer Interaction (INTERACT '05), Rome, Italy, pp. 823-834, 2005.
[4] Y. Tao, "Toward Computer-Aided Usability Evaluation for Evolving Interactive Software." Proceedings of the International Workshop on *Reflection, AOP and Meta-Data for Software Evolution, the* 21st European Conference on Object-Oriented Programming (ECOOP '07), Berlin, Germany, 2007.
[5] S. Bateman et al., "Interactive Usability Instrumentation," Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '09), Pittsburgh, USA, pp. 45-54, 2009.
[6] A. M. Tarja and G. S. Moldovan, "Automatic Usability Evaluation Using AOP," Proceedings of 2006 IEEE International Conference on Automation, Quality and Testing, Robotics, Los Alamitos, pp. 84-89, 2006.
[7] J. Tarby et al., "Trace-Based Usability Evaluation Using Aspect-Oriented Programming and Agent-Based Software Architecture," Human-Centered Software Engineering, Human-Computer Interaction Series, Vol. 3, Springer, pp. 257-276, 2009.
[8] A. Shekh and S. Tyerman, "An Aspect-Oriented Framework for Event Capture and Usability Evaluation," Communications in Computer and Information Science: Evaluation of Novel Approaches to Software Engineering, Vol. 69, Springer, pp. 107-119, 2010.
[9] R. Miles, "AspectJ Cookbook," O'Reilly Media, Inc., 2005.
[10] J. H. Kim et al., "Tracking Real-Time User Experience (TRUE): A Comprehensive Instrumentation Solution for Complex Systems," Proceedings of the 27th International Conference on Human Factors in Computing Systems (CHI '08), ACM, pp. 443-451, 2008.
[11] R. J. Torres, "Practitioner's Handbook for User Interface Design and Development," Prentice-Hall, 2002.
[12] NetBeans IDE, http://netbeans.org/.

DOI: 01.IJCSI.03.01.41